CHAPTER

ELEVEN

ACTORS

The Actor model takes the theme of object-oriented computation seriously and to an extreme. In an Actor system, everything is an actor (object). Actors communicate by sending each other messages. These messages are themselves actors.

Carl Hewitt and his colleagues at M.I.T. are developing the Actor model. Their work is devoted to refining the Actor concept, providing a formal semantics for Actor computations, and building programming systems that embody Actor principles. Our discussion of Actors focuses on two of these issues. Our first concern is the Actor metaphor—the philosophical consequences of treating all parts of a programming system uniformly. The remainder of this section deals with the semantics and implementation of Actor systems.

Data Abstraction

In Section 2-2 we introduced the idea of data abstraction—that the logical appearance of a program's data can be made independent of its physical implementation. Most abstraction mechanisms are designed for manipulating general data structures such as stacks, queues, and trees. Within each abstract object, conventional programming techniques are used to describe its behavior. The Actor metaphor extends the idea of abstraction to assert that all programming constructs are objects (actors), be they as simple as "the number 5," as functional as "factorial," or as complex as "this intelligent program."

Actor Theory

There are three kinds of actors: primitive actors, unserialized actors, and serialized actors. *Primitive actors* correspond to the data and procedure primitives of the computer system. For example, integer 5 and function + are primitive actors. Nonprimitive actors combine state and procedure. Serialized actors differ from unserialized actors in that *serialized actors* have local state that the actor itself can change, while *unserialized actors* cannot change their local state. A typical unserialized actor is factorial. Factorial can be implemented in terms of other primitive and unserialized actors, such as true, 1, and, recursively, factorial. A serialized actor associates local storage (state) with function. A typical serialized actor is a register that remembers the last value sent it. Such an actor retains its states between message receptions. Serialized actors process messages serially—one at a time.

Actors communicate by sending each other messages. An actor with a task for another actor to perform composes a *message* describing that task and *sends* that message to the other actor, the *target* of the message. At some time in the computational future, the message *arrives* at the target. At some time after its arrival, the message is *accepted*; its processing commences. An actor's *behavior* describes which messages the actor accepts and how it responds to those messages. Of course, messages are themselves actors, created by the primitive **create-unserialized-actor** actor.

One important class of actors is continuations [Reynolds 72; Strachey 74]. (Hewitt calls these *customer* or *complaint box* actors.) *Continuations* are actors that are sent as part of a message and that are intended (in certain circumstances) to extend part of the computation. Typically, an actor sends the result of its processing to a continuation. Since continuations are actors and all parts of messages are actors, messages can naturally include multiple continuations — perhaps one continuation to be sent the results of an error-free computation, another to be informed of syntax errors, and a third to handle exceptional machine conditions. A primitive, machine-oriented way of thinking about continuations is as a program-counter address combined with a set of registers. Sending a message to a continuation is like jumping to that address—a flexible form of computed-GOTO. Unlike Fortran's archetypical computed-GOTO, the range of possible continuations does not have to be specified before execution. In practice, it is more common to compose a continuation immediately before sending a message than it is to use a predefined one. Continuations are so powerful a control operation that they have been called "the ultimate GOTO" [Steele 77].

Every actor has a *script* (program) and *acquaintances* (data, local storage). When a message arrives at an actor, the actor's script is applied to that message. For example, primitive actor 1 accepts messages like "add yourself to 3, and send the answer to actor G0042." The script for actor 1 includes programs for handling addition messages, subtraction messages, multiplication messages,

equality messages, and printing messages. The script for a cell actor would include programs for handling value-setting messages, value-getting messages, and printing messages.

Primitive actors represent the primitive data objects of an actor system. Typical primitive actors include the integers and booleans. Each of primitive actors true and false has an important line in its script that handles messages of the form "if you are true then respond with thenval otherwise respond with elseval sending the answer to continuation." When actor true accepts such a message, it sends thenval to continuation; actor false sends elseval to continuation. That is, there is no conditional function. Instead, actors true and false know how to handle messages keyed to if. This parallels the lambda calculus, where true and false are functions (Section 1-2).

Unserialized actors are simply descriptions of functions. They compute their values by sending messages (with the right continuations) to other actors. Serialized actors are more powerful. They have both program (script) and local storage (acquaintances). When a serialized actor accepts a message, it becomes locked against further messages. One thing a locked serialized actor can do is *become* another actor. It unlocks in the process of "becoming." For example, a cell (register) actor might respond to messages of the form get with (actor) 3. On accepting a message telling it to set itself to 5, it locks. Its script then tells it to become the actor that responds to get messages with 5. The act of becoming this new actor unlocks the cell. The underlying system ensures that messages do not reach a locked actor. The locking of an actor on receipt of a message causes messages to be processed one at a time, that is, serially. Serialized actors do not behave as functions (in the mathematical sense)—they do not always respond to the same message with the same behavior. In a typical implementation, the underlying actor system would keep a queue of the messages that have arrived at a serialized actor. Hardware arbiters would determine an ordering on this queue. In Actor systems (just as in any postal service), the order in which two messages were sent is not necessarily the same as the order in which they arrive.

Serialized actors are created by sending a message to actor **create-serialized-actor**. This is equivalent to process creation. Sending a message to an actor is process activation. Concurrency in an Actor system arises when, after receiving a message, an actor sends several messages. Concurrency is reduced by an actor that receives a message and does not send any others.

Actor theory has several other rules for actor behavior. All actors have unique names. Actors start processing when they receive a message. Actor languages preclude iteration. Instead, repetition is achieved by having actors send themselves messages or by including themselves in the continuations of their messages. This last restriction has three significant consequences: (1) the evaluation of the code of any actor is finite; (2) the underlying system has the opportunity to interleave the evaluations of different actors and different messages; and (3) no message reception generates an unbounded (infinite) number of messages. Actor systems

(1) If an event E_1 precedes an event E_2, then only a finite number of events occurred between them.

(2) No event can immediately cause more than a finite number of events.

(3) Each event is generated by the sending of at most a single message.

(4) Of any two messages sent to an actor, one precedes the other in arrival order.

(5) The event ordering is well-founded (one can always take a finite number of "steps" back to the initial event).

(6) Only a finite number of actors can be created as an immediate result of a single event.

(7) An actor has a finite number of acquaintances at any time. Acquaintances are the actors to which it can send messages. An actor's acquaintances when processing a message are the union of its own acquaintances and the acquaintances of the message.

Figure 11-1 Laws for Actor systems.

do not guarantee the "prompt" delivery of messages, only their eventual delivery and processing. Thus, Actors support weak fairness.

Hewitt and Baker [Hewitt 77b] call the acceptance of a message by an actor an *event*. The intended semantics of Actor systems can be better understood by examination of their laws for Actor systems, given in Figure 11-1. These laws are meant to restrict Actor systems to those that can be physically implemented. The general goal of the laws is to ensure that Actor systems can be simulated by finite permutation of primitive events. The first, second, third, fifth, and sixth laws preclude possible loopholes to the finite permutation rules. The fourth law precludes simultaneity in Actor systems. The seventh law asserts that the physical storage of an actor is always bounded (though nothing in the Actor laws precludes that storage from growing during processing).

An important goal of the Actor work is universality. The Actor model is intended to be able to model all aspects of a computing system's behavior, from programs through processors and peripheral devices.

Actor terminology is confusing. There is a plethora of new names (such as "actor," "message," "target," "behavior," "script," and "event") for concepts that border on the familiar. The mystery is compounded by the assertion that most of these are members of a single class of object, actors, which all obey the same laws. In reality, the Actor metaphor is not as mysterious as the variety of names would imply. Lisp programmers have long recognized that the same structure can be put to many uses. A particular collection of cells and pointers can serve as a static data structure, as an argument list for a function, or as the code of the function itself. These are all bound together through the common denominator of the cons cell—each is a type of cons cell structure. The Actor metaphor is similar. When it asserts that "a message is also an actor," it is arguing the Actor equivalent of "a function's argument list is built using the Actor equivalent of cons."

Actor Practice

The Actor metaphor is attractive. It promises a uniformity of abstraction that is useful for real programming. One hopes that Actors will provide a release from the atomicity of conventional programming systems. The programmer need not determine the implementation of an object during design, only the form of its messages. No variable or function is so fine that it cannot be dynamically modified to exhibit a different behavior. The granularity of the processing elements in an Actor system is small enough to trace changes in the value of a single cell.

Hewitt and his colleagues have implemented several languages based on actors. The first was called Plasma; current systems include Act-1, Act-2, Omega, and Ether (we discuss Ether in Section 18-3). They developed these languages in a Lisp-based programming environment; their development drew on themes from the lambda calculus, Lisp, and artificial intelligence. The notions of functions as objects, lambda expressions, and continuations come from the lambda calculus. The idea of implementing actors as serialized *closures*—i.e., function code within a specified environment waiting to be applied to (accept) a message, is derived from Lisp.* And the concept of *pattern-matched invocation*—that the script of an actor should be described as a set of patterns instead of a sequential program—originated in artificial intelligence. Strangely enough, since pattern matching is simply another syntax for conditional expressions, the resulting system is remarkably similar to an applicative-order (call-by-value) lambda calculus interpreter (with processes) in Lisp. It is important to ensure in an Actor-like interpreter that the closures are correctly scoped.

Actors and the Lambda Calculus

What is the structure of an Actor implementation? In their description of the programming language Scheme, Sussman and Steele write [Sussman 75, p. 39]:

> This work developed out of an initial attempt to understand the actorness of Actors. Steele thought he understood it, but couldn't explain it; Sussman suggested the experimental approach of actually building an "Actors interpreter." This interpreter attempted to intermix the use of actors and Lisp lambda expressions in a clean manner. When it was completed, we discovered that the "actors" and the lambda-expressions were identical in implementation. Once we had discovered this, all the rest fell into place, and it was only natural to begin thinking about Actors in terms of lambda calculus.

The system that resulted from Sussman and Steele's work was the programming language Scheme—a language that is almost a direct implementation of the lambda calculus (with assignment statements) in Lisp (using lists and atoms

* In the past, closures have gone by the name "funargs" in the Lisp community. The "thunks" used to implement call-by-name in languages such as Algol 60 are also versions of closures [Ingerman 61].

instead of the strings of the pure lambda calculus). Scheme also relies heavily on closures. From a Scheme perspective, an actor's script is the closure expression and the set of an actor's acquaintances is an environment. In fact, Scheme is not Actors; even though Scheme can execute code isomorphic to Actor programs, the system does not address the Actor concern for weak fairness.

In Actors, side effects are formalized and given a specific interpretation by primitive serializers. This is particularly important in a distributed environment, where the use of multiple sites obscures the notion of simultaneous state. Steele discusses the mapping between lexically scoped Lisp and Actors in his paper, "Lambda, the Ultimate Declarative" [Steele 76].

Actor Language Features

Hewitt and his colleagues have extended Plasma and Act-1 with several features. They have primitives for delaying the processing of a message that creates an actor until a message is sent to it (**delay**) and for running several actors concurrently, accepting the answer of whichever terminates first (**race**). Act-1 scripts can be written so that message reception is done by pattern matching, instead of sequential conditional expressions. They have also implemented the "description system" Omega [Hewitt 80] that performs pattern matching and type checking based on simple inference mechanisms.

Certain assumptions underlie the implementation of any Actor system. One important concern is the recycling of resources. Actor systems assume the existence of a garbage collector that finds and reclaims the storage of inaccessible actors (as a Lisp system garbage collects inaccessible `cons` cells). The Actor metaphor also has no explicit notion of time and no facility (except complaint handlers) for handling communication failure. Instead, each actor must trust the underlying system to ensure that all communications are faithfully delivered.

Hewitt and his students have developed the formal semantics of Actor computations. Greif proved results about expressing Actor computations in terms of events and behaviors [Greif 75] and Clinger defined the semantics of Actor computations using power domains [Clinger 81].

Examples

Act-1 is a programming language with a well-defined and complicated syntax. Some of its syntactic goals are intended to aid artificial intelligence research, others to provide helpful functions for the system user. Act-1 is primarily a synthesis of continuation passing, function application, and data abstraction. To illustrate these ideas, we have chosen our own syntax; the commentary clarifies those uses that are unfamiliar programming constructs. We give an example of a syntactically correct, Act-1 program at the end of this section. Act-1's syntax is characterized by its Lisp-like use of parentheses and its keyword-oriented pattern

matching. In our examples, we preserve some of the keywords but remove many of the parentheses.

Factorial Our first example presents an unserialized actor, factorial (adapted from Hewitt [Hewitt 77a]). As actors have no loops, factorial is restricted to iteration by recursion—sending a message to itself. The message factorial sends itself has a continuation that embeds the original continuation sent the actor inside the action it performs. Our pseudoactor language is a mixture of Algol, Lisp, and the lambda calculus.

```
factorial ≡ λm.                    -- The factorial actor takes a message m.
  match m ⟨n c⟩                    -- Pattern-match m with a number n and a
                                      continuation c.
    if n = 1
        then (send c ⟨1⟩)          -- If n is 1, then send 1 to c. Send is similar to
                                      Lisp's apply, except that in actors, the
                                      matching is explicit.
    else
    if n > 1
        then (send factorial ⟨(n − 1) (λk.(send c ⟨n * k⟩))⟩)
                                   -- Otherwise, send factorial a message composed
                                      of n−1 and a continuation which will multiply
                                      n by the result of that factorial, and send the
                                      result to c.
```

An actor wishing to have the factorial of 3 computed and the resulting answer sent to continuation wantsanswer would invoke

```
(send factorial ⟨3 wantsanswer⟩)
```

This would be successively transformed to (we have primed the successive bound variables for clarity)

```
(send factorial ⟨2 (λk.(send wantsanswer ⟨3 * k⟩))⟩)

(send factorial ⟨1 (λk′.(send (λk.(send wantsanswer ⟨3 * k⟩))
                              ⟨2 * k′⟩))⟩)
```

When factorial accepts a message whose integer part is 1, it sends 1 to the continuation.

```
(send (λk′.(send (λk.(send wantsanswer ⟨3 * k⟩))
                 ⟨2 * k′⟩))
      ⟨1⟩)
```

Applying that continuation to ⟨1⟩ replaces all occurrences of k′ bound by the outer λk′ with 1 yielding

(**send** (λk.(**send** wantsanswer ⟨3 * k⟩)) ⟨2 * 1⟩)

Two times one is, of course, two. Sending that value to this continuation produces the result

(**send** wantsanswer ⟨3 * 2⟩)

(**send** wantsanswer ⟨6⟩)

Hence, actor wantsanswer is to be sent the message whose only element is 6. Fortunately, 3! = 6. The factorial actor computes iteratively by passing continuations. This solution avoids the arbitrarily deep stack required by a recursive solution. Instead, it creates an arbitrarily complex continuation. (We have simplified the program by treating multiplication and conditionals as primitive, instead of explicitly detailing their expansion by the primitive actors.)

Bank account A serialized actor can have permanent storage. We present an example of a part of a banking system, a serialized actor called an account [Hewitt 79]. An account has one permanent storage field, its balance. An account actor responds to deposit messages that add to its balance and to withdrawal messages that try to decrease it. The account actor bounces withdrawals that would leave it with negative funds.

```
account [balance] ≡ λm.
  match m ⟨"withdrawal"      -- A message whose first element is the word
                                "withdrawal"
           n                 -- second a number n
           c⟩                -- and third a continuation c.
    if balance > n then      -- If the balance is sufficient to cover this
                                withdrawal,
       parbegin              -- do these things in parallel:
          (send c ⟨"transaction_completed"⟩)
                             -- send an acknowledgment
          (become account (balance−n))
                             -- and transform into an account with balance of
                                (balance−n).
       parend
    else                                  -- If there are insufficient funds
       parbegin
          (send c ⟨"overdraft"⟩)
          (become account balance)        -- perform an identity
                                             transformation.
       parend;
```

```
match m ⟨"deposit"      -- If the message matches with a message whose first
                           element is deposit
        n               -- second is a number n
        c⟩              -- and third is a continuation c, then accept the
                           deposit.
    parbegin
        (send c ⟨"transaction_completed"⟩)
        (become account (balance + n))
                        -- This actor becomes one whose new balance is
                           (balance+n).
    parend
```

This actor responds to two kinds of messages: Messages of the form ⟨"withdrawal" n c⟩ request that n units be withdrawn from the account, and a confirmation of this action sent to c. The behavior of the actor depends on the balance of the account and the size of the withdrawal. Messages of the form ⟨"deposit" n c⟩ increase the balance by n, and send a confirmation to c. In either case, (serialized) actor account is transformed into an account with the new balance.* Actor account, written in Act-1 (from [Hewitt 82]), is as follows:

```
(defaction (new account (with balance =b))
    (create
        (is-request (a deposit (with amount =n)) do
            (become (new account (with balance (+ b n))))
            (reply (a deposit-receipt (with amount n))))
        (is-request (a withdrawal (with amount =n)) do
            (if (< b n)
                (then do (complain (an overdraft)))
                (else do
                    (become (new account (with balance (− b n))))
                    (reply (a withdrawal-receipt (with amount n))))))
        (is-request (a balance) do (reply b))))
```

Perspective

The Actor metaphor provides uniform, independent entities that communicate through message passing and continuations. This is a powerful and rewarding theme. In Section 18-3 we touch on some control organizations that conform to this model.

* One creates an account actor by sending a request to create_serialized_actor, the system storage allocation function.

PROBLEMS

11-1 Design an airline reservation system using the Actors model. Have your system keep a waiting list of passengers denied reservations. Inform the appropriate waiting passengers on cancellations.

11-2 Demonstrate how to write standard control structures such as **while**, **repeat**, and loops with multiple-level exits using Actors.

11-3 Write the program of an actor that behaves as a two-field cell, responding to messages that set and return the values of each field. Such an actor is similar to a Lisp cons cell.

REFERENCES

[**Clinger 81**] Clinger, W., "Foundations of Actor Semantics," Ph.D. dissertation, M.I.T., Cambridge, Massachusetts (May 1981). Also available as Technical Report 633, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts. Clinger gives a denotational semantics for an Actors-like system. His semantics relies on powerdomains. The difficult part of the semantics is combining indeterminacy and weak fairness.

[**Greif 75**] Greif, I., and C. E. Hewitt, "Actor Semantics of Planner-73," *Conf. Rec. 2d ACM Symp. Princ. Program. Lang.*, Palo Alto, California (January 1975), pp. 67–77. This paper describes actor semantics in terms of events and behaviors. Greif and Hewitt justify the use of side effects by arguing that programs with side effects are much more efficient than programs written in a pure, side-effect-free style. However, their argument assumes that computer systems are built with conventional von Neumann architectures.

[**Hewitt 77a**] Hewitt C. E., "Viewing Control Structures as Patterns of Passing Messages," *Artif. Intell.*, vol. 8, no. 3 (June 1977), pp. 323–364. This is one of the most applications-oriented of the Actors papers. Unfortunately, the programming examples are written in Plasma, which is even more obscure than its successor, Act-1.

[**Hewitt 77b**] Hewitt, C. E., and H. Baker, "Laws for Communicating Parallel Processes," in B. Gilchrist (ed.), *Information Processing 77: Proceedings of the IFIP Congress 77*, North Holland, Amsterdam (1977), pp. 987–992. Hewitt and Baker give a set of descriptive laws for the behavior of message-passing systems.

[**Hewitt 79**] Hewitt, C. E., G. Attardi, and H. Lieberman, "Specifying and Proving Properties of Guardians for Distributed Systems," in G. Kahn (ed.), *Semantics of Concurrent Computation*, Lecture Notes in Computer Science 70, Springer-Verlag, New York (1979), pp. 316–336. Hewitt et al. present examples of Actor programs for a checking account and a hardcopy server.

[**Hewitt 80**] Hewitt, C. E., G. Attardi, and M. Simi, "Knowledge Embedding in the Description System Omega," *Proc. 1st Annu. Natl. Conf. Artif. Intell.*, Stanford, California (August 1980), pp. 157–164. Omega is a knowledge representation language that unifies ω-order quantification calculus, type theory, and pattern matching within the Actor metaphor.

[**Hewitt 82**] Hewitt, C. E., personal communication, 1982.

[**Ingerman 61**] Ingerman, P., "Thunks," *CACM*, vol. 4, no. 1 (January 1961), pp. 55–58.

[**Reynolds 72**] Reynolds, J. C., "Definitional Interpreters for Higher-Order Programming Languages," *Proc. 25th ACM Natl. Conf.*, Boston (1972), pp. 717–740. Reynolds demonstrates a sequence of interpreters, progressing to an interpreter that uses continuations. He calls such continuations "escapes."

[**Steele 76**] Steele, G. L., Jr., "Lambda: The Ultimate Declarative," Memo 379, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts (November 1976). The third

section of this paper is a detailed comparison of a continuation-based Lisp programming language, Scheme, and an Actor language, Plasma.

[**Steele 77**] Steele, G. L., Jr., "Debunking the 'Expensive Procedure Call' Myth," *Proc. 30th ACM Natl. Conf.*, Seattle (October 1977), pp. 153–162. Revised as "Debunking the 'Expensive Procedure Call' Myth or, Procedure Call Implementations Considered Harmful or, Lambda: The Ultimate GOTO," Memo 443, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts, October 1977. This paper demonstrates (among other things) the power of lambda expressions (and continuations) as a programming tool.

[**Steele 78**] Steele, G. L., Jr., and G. J. Sussman, "The Revised Report on SCHEME, a Dialect of LISP," Memo 452, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts (January 1978).

[**Strachey 74**] Strachey, C., and C. P. Wadsworth, "Continuations—a Mathematical Semantics for Handling Full Jumps," Technical Monograph TRG-11, Programming Research Group, Oxford University, Oxford, England (1974). Strachey and Wadsworth introduce continuations. A continuation is a function that represents the remainder of a computation. They show that continuations are a much more powerful control structure than **goto**s.

[**Sussman 75**] Sussman, G. J., and G. L. Steele, Jr., "SCHEME: An Interpreter for Extended Lambda Calculus," Memo 349, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts (December 1975). This report documents an attempt to implement Actor concepts. Included in the report is the code for a Scheme interpreter written in Lisp. While Scheme does not deal with every issue of Actor systems, it captures many of the important concepts in a particularly clean fashion.